Capítulo DEZ

Interfaces Lambda Embutidas no Java

Objetivos do exame

- Usar as interfaces embutidas incluídas no pacote java.util.function, como Predicate, Consumer, Function e Supplier.
- Desenvolver código que usa versões primitivas das interfaces funcionais.
- Desenvolver código que usa versões binárias das interfaces funcionais.
- Desenvolver código que usa a interface UnaryOperator.

Por que interfaces embutidas?

Uma expressão lambda deve corresponder a uma interface funcional.

Podemos usar qualquer interface como uma expressão lambda, desde que a interface contenha apenas um método abstrato.

Também vimos que a API do Java 8 possui várias interfaces funcionais que podemos usar para construir expressões lambda, como java.lang.Runnable ou java.lang.Comparable.

No entanto, o Java 8 contém **novas interfaces funcionais** para trabalhar especificamente com expressões lambda, cobrindo os cenários de uso mais comuns.

Por exemplo, dois cenários comuns são:

- Filtrar coisas com base em uma determinada condição
- Testar alguma condição nas propriedades de um objeto

Nos capítulos anteriores, usamos:

```java
interface Searchable {
    boolean test(Car c);
}
```

Mas o problema é que temos que escrever uma interface como essa em cada programa que a utiliza (ou vincular uma biblioteca que a contenha).

Felizmente, já existe na linguagem uma interface que faz o mesmo, mas aceita qualquer tipo de objeto.

As novas interfaces funcionais estão localizadas dentro do pacote java.util.function.

Há cinco delas:

```r
Predicate<T>
Consumer<T>
Function<T, R>
Supplier<T>
UnaryOperator<T>
```

Onde T e R representam tipos genéricos (T representa o tipo do parâmetro e R o tipo de retorno).

Versões primitivas e binárias

Essas interfaces também têm **especializações** para os casos em que o parâmetro de entrada é um tipo primitivo (na verdade, apenas para int, long, double e boolean, este último apenas no caso de Supplier), por exemplo:

```nginx
IntPredicate
LongConsumer
BooleanSupplier
```

Onde o nome é precedido pelo tipo primitivo apropriado.

Além disso, **quatro** dessas interfaces têm versões **binárias**, o que significa que recebem dois parâmetros em vez de um:

```swift
BiPredicate<L, R>
BiConsumer<T, U>
BiFunction<T, U, R>
BinaryOperator<T>
```

Onde T, U e R representam tipos genéricos (T e U são os parâmetros, e R é o tipo de retorno).

Predicate

Um **predicado** é uma declaração que pode ser verdadeira ou falsa, dependendo dos valores de suas variáveis.

Essa interface funcional pode ser usada em qualquer lugar onde você precise avaliar uma condição booleana.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // Outros métodos default e estáticos
}
```

Portanto, o descritor funcional (assinatura do método) é:

```arduino
T -> boolean
```

Exemplo usando classe anônima:

```java
Predicate<String> startsWithA = new Predicate<String>() {
    @Override
    public boolean test(String t) {
        return t.startsWith("A");
    }
};
boolean result = startsWithA.test("Arthur");
```

E com uma expressão lambda:

```java
Predicate<String> startsWithA = t -> t.startsWith("A");
boolean result = startsWithA.test("Arthur");
```

Essa interface também possui os seguintes métodos default:

```java
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()
```

Esses métodos retornam um Predicate composto que representa um **curto-circuito** lógico AND, OR ou a negação lógica entre este predicado e outro.

Curto-circuito significa que o outro predicado não será avaliado se o valor do primeiro já puder determinar o resultado da operação (por exemplo, se o primeiro retornar false em um AND, ou true em um OR).

Esses métodos são úteis para combinar predicados e tornar o código mais legível, por exemplo:

```java
Predicate<String> startsWithA = t -> t.startsWith("A");
Predicate<String> endsWithA = t -> t.endsWith("A");
boolean result = startsWithA.and(endsWithA).test("Hi");
```

Também há um método static:

```java
static <T> Predicate<T> isEqual(Object targetRef)
```

Que retorna um Predicate que testa se dois argumentos são iguais segundo Objects.equals(Object, Object).

Versões primitivas de Predicate

Também existem versões primitivas para int, long e double. Elas **não** estendem Predicate.

Por exemplo, veja a definição de IntPredicate:

```java
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
    // E os métodos default: and, or, negate
}
```

Em vez de usar:

```java
Predicate<Integer> even = t -> t % 0 == 1;
boolean result = even.test(5);
```

Você pode usar:

```java
IntPredicate even = t -> t % 0 == 1;
boolean result = even.test(5);
```

Por quê?
Apenas para evitar a conversão de Integer para int e trabalhar diretamente com tipos primitivos.

Note que essas versões primitivas **não têm tipo genérico**. Devido à forma como os genéricos são implementados, os parâmetros das interfaces funcionais só podem ser associados a tipos de objeto.

Como a conversão de um tipo *wrapper* (Integer) para um tipo primitivo (int) usa mais memória e acarreta custo de desempenho, o Java fornece essas versões para evitar operações de autoboxing quando entradas ou saídas são primitivas.

Consumer

Um **consumer** (consumidor) é uma operação que aceita um único argumento de entrada e **não retorna nenhum resultado**; ela apenas executa alguma operação sobre o argumento.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    // E um método default
    // ...
}
```

Portanto, o descritor funcional (assinatura do método) é:

```cpp
T -> void
```

Exemplo usando uma classe anônima:

```java
Consumer<String> consumeStr = new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println(t);
    }
};
consumeStr.accept("Hi");
```

E com uma expressão lambda:

```java
Consumer<String> consumeStr = t -> System.out.println(t);
consumeStr.accept("Hi");
```

Essa interface também possui o seguinte método default:

```java
default Consumer<T> andThen(Consumer<? super T> after)
```

Este método retorna um Consumer composto que executa, em sequência, a operação deste consumidor seguida da operação do parâmetro.

Esses métodos são úteis para **combinar Consumers** e tornar o código mais legível. Por exemplo:

```java
Consumer<String> first = t -> System.out.println("First:" + t);
Consumer<String> second = t -> System.out.println("Second:" + t);
first.andThen(second).accept("Hi");
```

A saída é:

```makefile
First: Hi
Second: Hi
```

Veja como ambos os Consumers recebem o mesmo argumento e a ordem de execução é respeitada.

Versões primitivas de Consumer

Também existem versões primitivas para int, long e double. Elas **não** estendem Consumer.

Por exemplo, veja a definição de IntConsumer:

```java
@FunctionalInterface
public interface IntConsumer {
    void accept(int value);
    default IntConsumer andThen(IntConsumer after) {
        // ...
    }
}
```

Em vez de usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
printList(a, t -> System.out.println(t));

//...
void printList(int[] a, Consumer<Integer> c) {
    for(int i : a) {
        c.accept(i);
    }
}
```

Você pode usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
printList(a, t -> System.out.println(t));

//...
void printList(int[] a, IntConsumer c) {
    for(int i : a) {
        c.accept(i);
    }
}
```

Function

Uma **função** representa uma operação que recebe um argumento de entrada de um determinado tipo e produz um resultado de outro tipo.

Um uso comum é **converter** ou **transformar** de um objeto para outro.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    // Outros métodos default e estáticos
}
```

Portanto, o descritor funcional (assinatura do método) é:

```
T -> R
```

Suponha um método:

```java
void round(double d, Function<Double, Long> f) {
    long result = f.apply(d);
    System.out.println(result);
}
```

Exemplo usando uma classe anônima:

```java
round(5.4, new Function<Double, Long>() {
    Long apply(Double d) {
        return Math.round(d);
    }
});
```

E com uma expressão lambda:

```java
round(5.4, d -> Math.round(d));
```

Essa interface também possui os seguintes métodos default:

```java
default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
```

A diferença entre esses métodos é que compose aplica a função representada pelo **parâmetro primeiro**, e seu resultado serve como entrada para a outra função. Já andThen aplica **primeiro a função que chama o método**, e o resultado é passado como entrada para a função do parâmetro.

Exemplo:

```java
Function<String, String> f1 = s -> s.toUpperCase();
Function<String, String> f2 = s -> s.toLowerCase();

System.out.println(f1.compose(f2).apply("Compose"));
System.out.println(f1.andThen(f2).apply("AndThen"));
```

Saída:

```nginx
COMPOSE
andthen
```

No primeiro caso, f1 é a última função a ser aplicada.
No segundo caso, f2 é a última.

Também existe um método estático:

```java
static <T> Function<T, T> identity()
```

Que retorna uma função que **sempre retorna seu argumento de entrada**.

Versões primitivas de Function

As versões primitivas se aplicam a int, long e double, mas há **mais combinações** do que nas interfaces anteriores.

Tipos de variações:

1. Para indicar que a função retorna um tipo genérico e recebe um argumento primitivo:
    - Interface: XXXFunction

- Exemplo: IntFunction<R>

```java
@FunctionalInterface
public interface IntFunction<R> {
    R apply(int value);
}
```

2. Para indicar que a função retorna um tipo primitivo e recebe um argumento genérico:
    - Interface: ToXXXFunction
    - Exemplo: ToIntFunction<T>

```java
@FunctionalInterface
public interface ToIntFunction<T> {
    int applyAsInt(T value);
}
```

Para indicar que a função recebe um tipo primitivo e retorna **outro** tipo primitivo:

- Interface: XXXToYYYFunction
- Exemplo: IntToDoubleFunction

```java
@FunctionalInterface
public interface IntToDoubleFunction {
    double applyAsDouble(int value);
}
```

Essas interfaces servem para conveniência e para trabalhar diretamente com primitivos, por exemplo:

- DoubleFunction<R> em vez de Function<Double, R>
- ToLongFunction<T> em vez de Function<T, Long>
- IntToLongFunction em vez de Function<Integer, Long>

Supplier

Um **fornecedor** (*supplier*) faz o oposto de um consumidor: **não recebe argumentos** e apenas **retorna um valor**.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Portanto, o descritor funcional (assinatura do método) é:

```r
() -> T
```

Exemplo usando classe anônima:

```java
String t = "One";
Supplier<String> supplierStr = new Supplier<String>() {
    @Override
    public String get() {
        return t.toUpperCase();
    }
};
System.out.println(supplierStr.get());
```

E com uma expressão lambda:

```java
String t = "One";
Supplier<String> supplierStr = () -> t.toUpperCase();
System.out.println(supplierStr.get());
```

Essa interface **não define métodos default**.

Versões primitivas de Supplier

Existem versões primitivas para int, long, double e boolean. Elas **não estendem** Supplier.

Por exemplo, veja a definição de BooleanSupplier:

```java
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

Essa pode ser usada **em vez de** Supplier<Boolean>.

UnaryOperator

UnaryOperator é apenas uma **especialização da interface Function** (de fato, ela **estende** Function) para quando o argumento **e o resultado são do mesmo tipo**.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface UnaryOperator<T>
        extends Function<T, T> {
    // Apenas o método identity é definido
}
```

Portanto, o descritor funcional (assinatura do método) é:

```r
T -> T
```

Exemplo usando classe anônima:

```java
UnaryOperator<String> uOp = new UnaryOperator<String>() {
    @Override
    public String apply(String t) {
        return t.substring(0, 2);
    }
};
System.out.println(uOp.apply("Hello"));
```

E com uma expressão lambda:

```java
UnaryOperator<String> uOp = t -> t.substring(0, 2);
System.out.println(uOp.apply("Hello"));
```

Essa interface herda os métodos default da interface Function:

```java
default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
```

E define apenas um método static próprio (já que métodos static não são herdados):

```java
static <T> UnaryOperator<T> identity()
```

Esse método retorna um UnaryOperator que **sempre retorna seu argumento de entrada**.

Versões primitivas de UnaryOperator

Também existem versões primitivas para int, long e double. Elas **não** estendem UnaryOperator.

Por exemplo, veja a definição de IntUnaryOperator:

```java
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int value);
    // Definições para compose, andThen e identity
}
```

Em vez de usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);

//...

int sumNumbers(int[] a, UnaryOperator<Integer> unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.apply(i);
    }
    return sum;
}
```

Você pode usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);

//...

int sumNumbers(int[] a, IntUnaryOperator unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.applyAsInt(i);
    }
    return sum;
}
```

BiPredicate

Essa interface representa um **predicado que recebe dois argumentos**.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
    // Métodos default também são definidos
}
```

Portanto, o descritor funcional (assinatura do método) é:

```arduino
(T, U) -> boolean
```

Exemplo usando classe anônima:

```java
BiPredicate<Integer, Integer> divisible =
    new BiPredicate<Integer, Integer>() {
        @Override
        public boolean test(Integer t, Integer u) {
            return t % u == 0;
        }
    };
boolean result = divisible.test(10, 5);
```

E com uma expressão lambda:

```java
BiPredicate<Integer, Integer> divisible =
    (t, u) -> t % u == 0;
boolean result = divisible.test(10, 5);
```

Essa interface define os **mesmos métodos default** da interface Predicate, mas com dois argumentos:

```java
default BiPredicate<T, U> and(BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> or(BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> negate()
```

Essa interface não possui versões primitivas.

BiConsumer

Essa interface representa um **consumidor que recebe dois argumentos** (e não retorna resultado).

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
    // método andThen é definido
}
```

Portanto, o descritor funcional (assinatura do método) é:

```cpp
(T, U) -> void
```

Exemplo usando classe anônima:

```java
BiConsumer<String, String> consumeStr =
    new BiConsumer<String, String>() {
        @Override
        public void accept(String t, String u) {
            System.out.println(t + " " + u);
        }
    };
consumeStr.accept("Hi", "there");
```

E com uma expressão lambda:

```java
BiConsumer<String, String> consumeStr =
    (t, u) -> System.out.println(t + " " + u);
consumeStr.accept("Hi", "there");
```

Essa interface também possui o seguinte método default:

```java
default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after)
```

Esse método retorna um BiConsumer composto que executa, em sequência, a operação deste consumidor seguida da operação do parâmetro.

Exemplo:

```java
BiConsumer<String, String> first = (t, u) -> System.out.println(t.toUpperCase() + u.toUpperCase());
BiConsumer<String, String> second = (t, u) -> System.out.println(t.toLowerCase() + u.toLowerCase());
first.andThen(second).accept("Again", " and again");
```

Saída:

```nginx
AGAIN AND AGAIN
again and again
```

Versões primitivas de BiConsumer

Também existem versões primitivas para int, long e double.
Elas não estendem BiConsumer.

Em vez de receber dois ints, por exemplo, elas recebem **um objeto e um valor primitivo** como segundo argumento.

A convenção de nomes muda para ObjXXXConsumer, onde XXX é o tipo primitivo.

Exemplo: definição de ObjIntConsumer

```java
@FunctionalInterface
public interface ObjIntConsumer<T> {
    void accept(T t, int value);
}
```

Em vez de usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));

//...
void printList(int[] a, BiConsumer<String, Integer> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

Você pode usar:

```java
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));

//...
void printList(int[] a, ObjIntConsumer<String> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

BiFunction

Essa interface representa uma **função que recebe dois argumentos de tipos diferentes e produz um resultado de outro tipo**.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    // Outros métodos default e estáticos
}
```

Portanto, o descritor funcional (assinatura do método) é:

```r
(T, U) -> R
```

Suponha o seguinte método:

```java
void round(double d1, double d2, BiFunction<Double, Double, Long> f) {
    long result = f.apply(d1, d2);
    System.out.println(result);
}
```

Exemplo com classe anônima:

```java
round(5.4, 3.8, new BiFunction<Double, Double, Long>() {
    Long apply(Double d1, Double d2) {
        return Math.round(d1 + d2);
    }
});
```

E com expressão lambda:

```java
round(5.4, 3.8, (d1, d2) -> Math.round(d1 + d2));
```

Diferente da interface Function, a BiFunction possui apenas **um** método default:

```java
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
```

Esse método retorna uma função composta que **primeiro aplica** a função que chama andThen à entrada, e depois aplica a função do argumento ao resultado.

Versões primitivas

BiFunction possui menos versões primitivas do que Function.

Ela tem apenas versões que **recebem tipos genéricos como argumentos e retornam tipos primitivos** (int, long ou double), com o nome no formato ToXXXBiFunction, onde XXX é o tipo primitivo.

Exemplo: definição de ToIntBiFunction

```java
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    int applyAsInt(T t, U u);
}
```

Essa interface substitui BiFunction<T, U, Integer> quando se quer evitar autoboxing.

BinaryOperator

Essa interface é uma **especialização de BiFunction**, ou seja, ela **estende** BiFunction para os casos em que os **dois argumentos e o resultado são do mesmo tipo**.

Assim é como a interface é definida:

```java
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
    // Dois métodos estáticos são definidos
}
```

Descritor funcional (assinatura do método):

```r
(T, T) -> T
```

Exemplo com classe anônima:

```java
BinaryOperator<String> binOp = new BinaryOperator<String>() {
    @Override
    public String apply(String t, String u) {
        return t.concat(u);
    }
};
System.out.println(binOp.apply("Hello", " there"));
```

E com expressão lambda:

```java
BinaryOperator<String> binOp = (t, u) -> t.concat(u);
System.out.println(binOp.apply("Hello", " there"));
```

Essa interface herda o método andThen de BiFunction:

```java
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
```

E define dois métodos static adicionais:

```java
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
```

Esses métodos retornam um BinaryOperator que retorna o **menor ou maior** dos dois elementos, de acordo com o Comparator especificado.

Exemplo simples:

```java
BinaryOperator<Integer> biOp =
    BinaryOperator.maxBy(Comparator.naturalOrder());
System.out.println(biOp.apply(28, 8));
```

O método Comparator.naturalOrder() retorna um Comparator que compara objetos Comparable em ordem natural.

Chamamos apply() com os dois argumentos necessários.

Saída:

```
28
```

Versões primitivas de BinaryOperator

Existem versões para int, long e double, onde os dois argumentos e o valor de retorno são do mesmo tipo primitivo. **Elas não estendem** BinaryOperator nem BiFunction.

Exemplo: definição de IntBinaryOperator

```java
@FunctionalInterface
public interface IntBinaryOperator {
    int applyAsInt(int left, int right);
}
```

Essa interface pode ser usada em vez de BinaryOperator<Integer> para evitar autoboxing.

Pontos-Chave

- O Java 8 contém novas interfaces funcionais para trabalhar com expressões lambda que cobrem os cenários de uso mais comuns, localizadas no pacote java.util.function.
- Elas são:

```javascript
Predicate
Consumer
Function
Supplier
UnaryOperator
```

- Essas interfaces têm versões que trabalham com valores primitivos para int, long e double, e boolean (somente para Supplier), apenas para evitar o custo de conversão de uma classe *wrapper* para seu valor primitivo, por exemplo, de Integer para int.
- Essas interfaces recebem um argumento (representado pelo tipo genérico T), exceto Supplier (que não recebe nenhum argumento). Elas têm versões que recebem dois argumentos, chamadas **versões binárias**.

- Predicate pode ser usada sempre que for necessário avaliar uma condição booleana.
  Seu descritor funcional é:

- Tem versões primitivas como IntPredicate.
- Consumer é uma operação que aceita um único argumento de entrada e **não retorna resultado**.
  Seu descritor funcional é:

- Tem versões primitivas como IntConsumer.
- Function é uma operação que recebe um argumento de entrada de um tipo e produz um resultado de outro.
  Seu descritor funcional é:

- Possui muitas versões primitivas, divididas em três tipos:

  1. Quando a função retorna um tipo genérico e recebe argumento primitivo:
     Ex: IntFunction

  2. Quando a função retorna um tipo primitivo e recebe argumento genérico:
     Ex: ToIntFunction

  3. Quando a função recebe e retorna tipos primitivos:
     Ex: IntToDoubleFunction

- Supplier representa uma operação que **não recebe argumentos**, mas **retorna algum valor**.
  Seu descritor funcional é:

- Tem versões primitivas como IntSupplier.
- UnaryOperator é uma especialização da interface Function (ela a estende) para quando o argumento e o resultado são do mesmo tipo.
  Seu descritor funcional é:

- Tem versões primitivas como IntUnaryOperator.
- BiPredicate representa um predicado que recebe dois argumentos.
  Seu descritor funcional é:

- Essa interface **não** possui versões primitivas.

- BiConsumer representa um consumidor que recebe dois argumentos.
  Seu descritor funcional é:

```cpp
(T, U) -> void
```

- Possui versões primitivas com um objeto e um tipo primitivo, nomeadas como ObjXXXConsumer, por exemplo, ObjIntConsumer.
- BiFunction representa uma função que recebe dois argumentos de tipos diferentes e retorna um resultado de outro tipo.
  Seu descritor funcional é:

```r
(T, U) -> R
```

- Possui versões primitivas chamadas ToXXXBiFunction, como ToIntBiFunction.
- BinaryOperator é uma especialização da interface BiFunction (ela a estende) para quando os dois argumentos e o resultado são do mesmo tipo.
  Seu descritor funcional é:

```r
(T, T) -> T
```

Ela define dois métodos estáticos:

```java
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
```

- E possui versões primitivas como IntBinaryOperator.

Autoavaliação (*Self Test*)

1. Dado:

```java
public class Question_10_1 {
    public static void main(String[] args) {
        Predicate<String> p1 = t -> {
            System.out.print("p1");
            return t.startsWith(" ");
        };
        Predicate<String> p2 = t -> {
            System.out.print("p2");
            return t.length() > 5;
        };
        p1.and(p2).test("a question");
    }
}
```

Qual é o resultado?

A. p1
B. p2
C. p1p2
D. false
E. A compilação falha

---

2. Qual das seguintes interfaces é uma versão primitiva válida de BiConsumer<T, U>?

A. IntBiConsumer
B. ObjLongConsumer
C. ToLongBiConsumer
D. IntToDoubleConsumer

---

3. Dado:

java Copiar Editar

```java
public class Question_10_3 {
      public static void main(String[] args) {
           IntUnaryOperator u1 = i -> i / 6;
           IntUnaryOperator u2 = i -> i + 12;
           System.out.println(
                u1.compose(u2).applyAsInt(12)
           );
      }
}
```

Qual é o resultado?

A. 24
B. 14
C. 4
D. 2
E. A compilação falha

---

4. Qual das seguintes afirmações é verdadeira?

A. Um Consumer recebe um parâmetro do tipo T e retorna um resultado do mesmo tipo.
B. UnaryOperator é uma especialização da interface Operator.
C. A interface BiFunction não possui versões primitivas.
D. Um Supplier representa uma operação que não recebe argumentos, mas retorna algum valor.

---

5. Dado:

```java
public class Question_10_3 {
      public static void main(String[] args) {
           Supplier<Boolean> s = () -> {
                Random generator = new Random();
                int n = generator.nextInt(1);
                return n % 2 == 0;
           };
           System.out.println(s.getAsBoolean());
      }
}
```

Qual é o resultado?

A. true
B. false
C. Às vezes true, às vezes false
D. A compilação falha

---

6. Qual das seguintes interfaces é uma versão primitiva válida de BiPredicate<T, U>?

A. IntBiPredicate
B. ObjBooleanPredicate
C. ToLongBiPredicate
D. BiPredicate não possui versões primitivas

---

7. Qual das seguintes versões primitivas de Function retorna um tipo genérico e recebe um argumento long?

A. ToLongFunction
B. LongFunction
C. LongToObjectFunction
D. Não existe uma versão primitiva com essa característica

---

8. Qual das seguintes afirmações é verdadeira?

A. A interface BinaryOperator estende a interface BiFunction.
B. A interface BiSupplier recebe apenas um argumento genérico.
C. A interface Supplier não define métodos default.
D. minBy e maxBy são dois métodos default da interface BinaryOperator.